Transactional Memory

SHULE LI

CS458

Outline

- Transaction: concepts and backgrounds
- Hardware Transactional Memory
  - Herlihy's 1993 Implementation
- Software Transactional Memory
  - DSTM
  - RSTM

Transaction

What is a transaction?

A unit of program execution that accesses and possibly updates various data items as an atomic operation.

Example transaction:

Transfer $50 from account A to B.

```
T:      read A;
        A = A – 50;
        write A;
        read B;
        B = B + 50;
        write B;
```

ACID

Atomicity: transactions are “all or none”. If a transaction fails, any change it made must be undone.

Consistency: a single transaction should preserve system invariants.

Isolation: although transactions can execute concurrently, the system ensures there appears to be a global observable order for all transactions (serializable).

Durability: once committed, the change is permanent even if system crashes.

Example transaction:

Transfer $50 from account A to B.

T:

```
read A;
A = A – 50;
write A;
read B;
B = B + 50;
write B;
```

Nonblocking Algorithms

Wait-Free: strongest requirement. All processors executing concurrently are able to make progress in a finite amount of steps. Rules out deadlock and starvation.

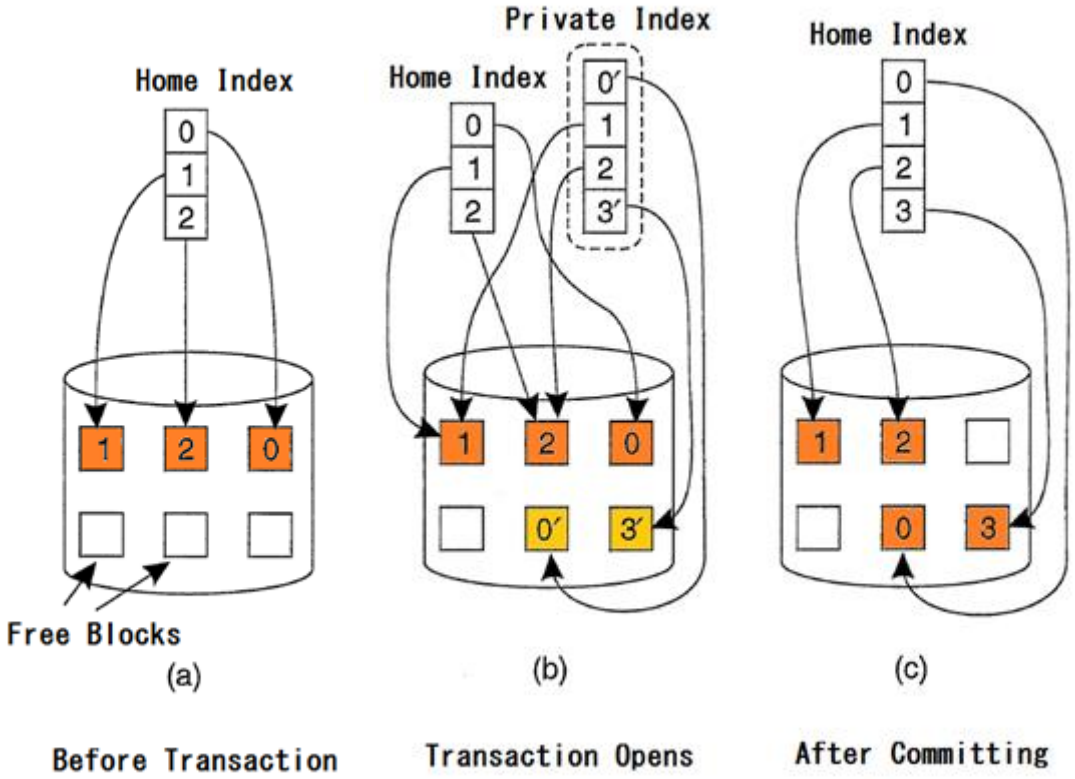
Lock-Free: medium requirement. At least one processor executing concurrently is able to make progress in a finite amount of steps. Rules out deadlock, but may have starvation.

Obstruction-Free: weakest requirement. A processor is able to progress in a finite amount of its own steps in the absence of contention. Rules out deadlock, but may have livelock.

How to Implement?

Private Workspace:

- Processor starting a transaction can copy all of the relevant data into a private workspace so that the shared memory locations are only modified when committing. Only need to copy indices on writes (shadow blocks).
- All of the other processes still refer to the old index when accessing the blocks.
- On commit, private index is merged with the home index, out of date blocks are freed. On abort, shadow blocks are freed and private index is deleted.



Atomicity and Durability

Writeahead Log

- Keep a log for system modifications of a certain transaction. Logs can be used to create checkpoints, redo or undo failed transactions.

Example transactions:

- T_0:Transfer $50 from account A to B.
- T_1: Withdraw $100 from account C.

T_0:
read A;
A = A – 50;
write A;
read B;
B = B + 50;
write B;

T_1:
read C;
C = C – 100;
write C;

Log:	System:
<T_0, start>	
<T_0, A, 1000, 950>	A = 950
<T_0, B, 2000, 2050>	B = 2050
<T_0, commit>	
<T_1, start>	
<T_1, C, 700, 600>	C = 600
<T_1, commit>	

Checkpoints and Rollbacks

- Checkpoints helps system recovery and rollbacks: only have to rollback to checkpoints.
- Rollbacks helps enforcing atomicity, consistency and durability. On rollback, scan the log backwards till the checkpoint. For each write, restore old values. In the example, C would be restored to 700. For each restoration, create a new “undo” log record.
- Once a record <T_i, start> is found, a new log record <T_i, abort> is appended.

Observed Log:

<T_0, start>

<T_0, A, 1000, 950>

<T_0, B, 2000, 2050>

<T_0, commit>

<checkpoint T_0>

<T_1, start>

<T_1, C, 700, 600>

system crash or abort request

Transactional Memory: Why use them?

A unit of program execution that accesses and possibly updates various data items as an atomic operation.

- Avoid using locks.
  - Priority Inversion
  - Lock Convoy (inefficiency on wake up)
  - Deadlock (loop of requests)
- Creates generic nonblocking algorithm.
- Hardware support as well as software implementations.

Priority inversion: low priority task holding the shared resource is preempted by an incoming medium priority unblocked task, holding back the high priority task requiring the same shared resource to wait indefinitely.

A Variety of TM

Hardware:

Supports transactions on the level of cache and bus protocols.

- Rock (cancelled by Oracle)
- BlueGene/Q

Software:

- Word-based
  - Shavit et al 1995 (first)
  - Hash table implementation (Harris et al, 2003)
  - TL2 (Dice et al 2006)
- Object based
  - Dynamic STM (Herlihy et al 2003)
  - OSTM (Fraser et al 2003)
  - RSTM (University of Rochester 2006)
  - DSTM2 (Sun Lab)
  - XTM (Microsoft Research)

Summary of Concepts

- Transaction: a set of operations that guarantee ACID.
- Implementing atomicity: private workspaces (make local copies before modification).
- Checkpoints and rollbacks: enhance atomicity and durability.
- Using nonblocking algorithms such as transactions help us avoiding traps in locks.
  - Priority inversion
  - Lock convoying
  - Deadlocks

HTM: A Simple Code Example

```
shared int counter;

void process(int work)
{
    int success = 0, backoff = BACKOFF_MIN;
    unsigned wait;
    while (success < work) {
        ST(&counter, LTX(&counter) + 1);
        if (COMMIT()) {
            success++;
            backoff = BACKOFF_MIN;
        } else {
            wait = random() % (01 << backoff);
            while (wait--);
            if (backoff < BACKOFF_MAX)
            backoff++;
        }
    }
}
```

- From Herlihy93. Transactions are used to increment a global counter.
- Exponential backoff is used when commit fails
- From this example we can see some important transaction primitives proposed.

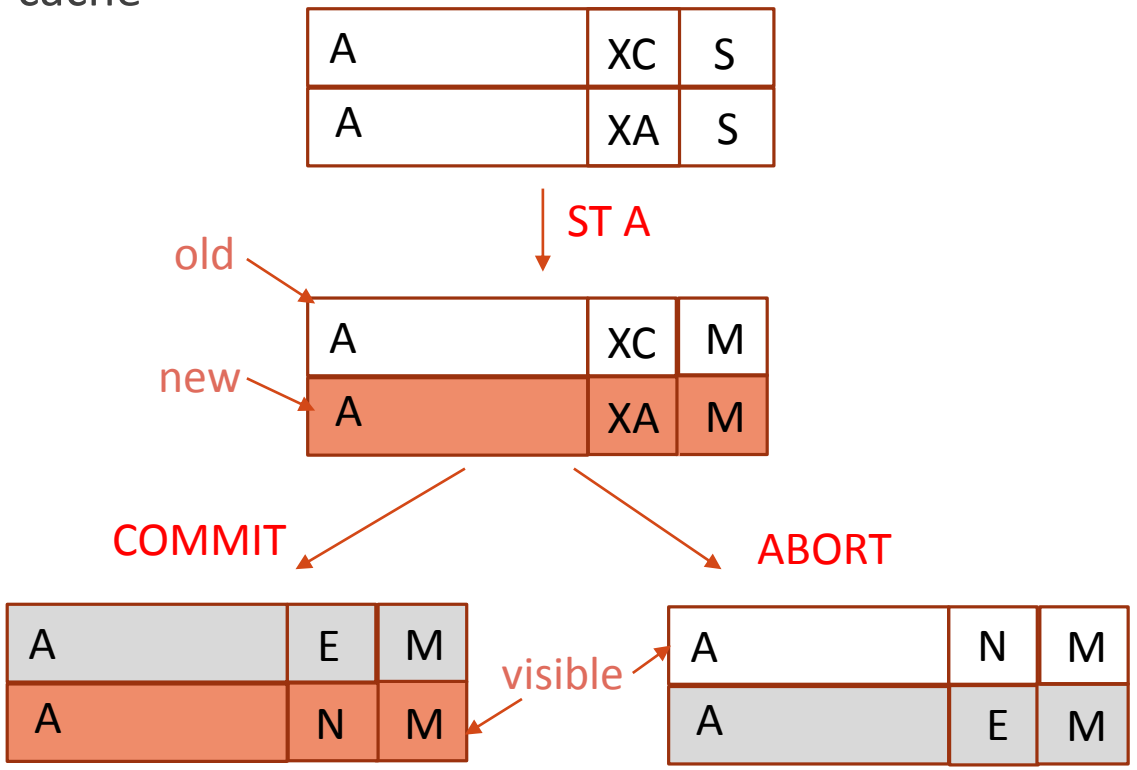
ST: store-transactional
LT: load-transactional
LTX: load-transactional-exclusive
COMMIT: make transaction permanent
ABORT: discard all writes
VALIDATE: check transaction's current status

Snoopy-based Protocol

- Two Caches: regular cache (directly-mapped) and transactional cache (fully-associative).
- The transactional cache works as a private workspace (modifications can be discarded on abort).
- Cache line states: MESI
- Cache lines in transactional cache has additional tags to label transactional states.

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

Example: bring cache line A into transactional cache



Bus Cycles

Name	Kind	Meaning	New access
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	both	write back	exclusive
T_READ	trans	read value	shared
T_RFO	trans	read value	exclusive
BUSY	trans	refuse access	unchanged

Snoop result handling of the transactional cache

TSTATUS = true: return BUSY

TSTATUS = false:

- Regular cycles: act as normal cache on lines tagged N. Ignore other lines.
- Transactional cycles: return BUSY except for TR on lines with state S.
- In general: transactional cache lines are made visible to other processors once they are tagged N.

Eviction Policy

On eviction, search for lines tagged with E, then N (need to WB if the line is in M), then XC (need to WB if the line is in M. why?).

Consider the following transactions (A = 0 initially):

LTX A

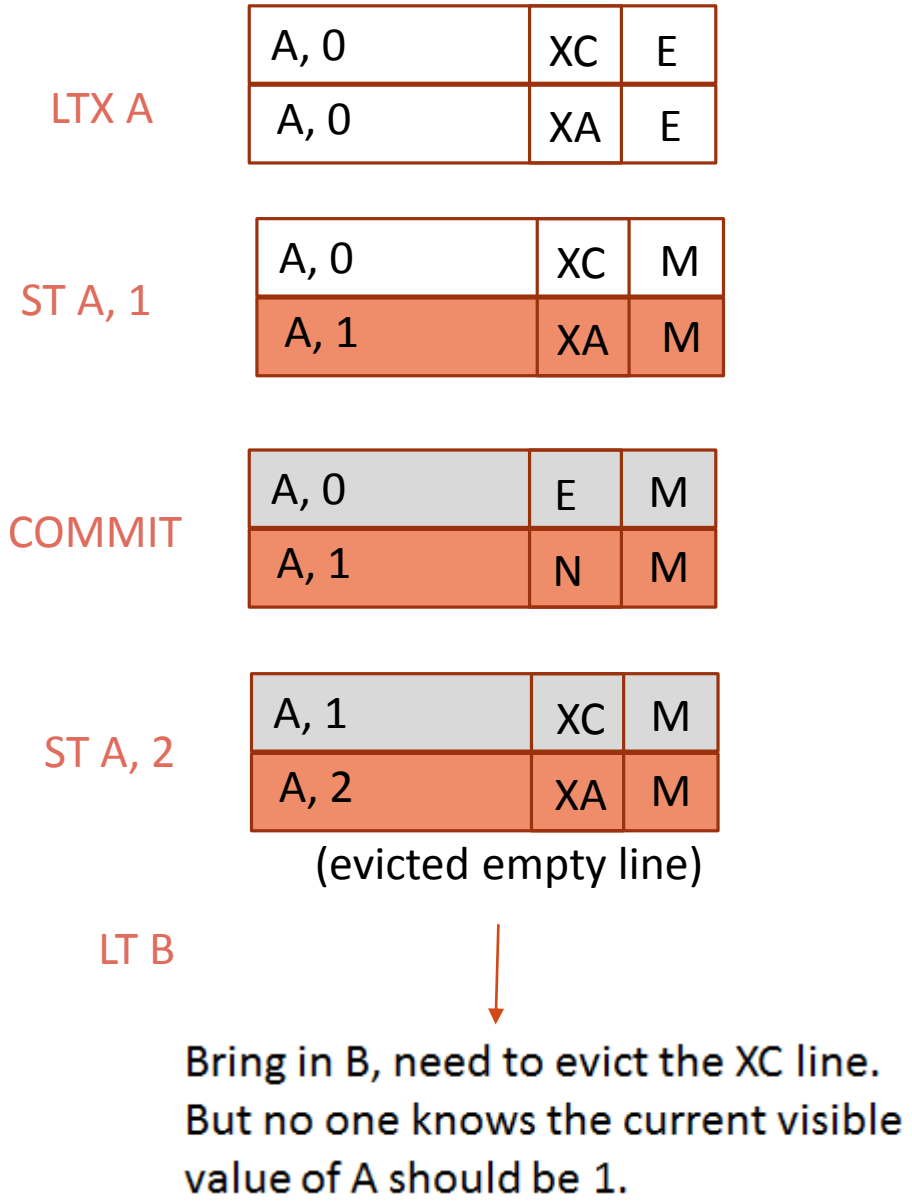
ST A, 1

COMMIT

ST A , 2

LT B

COMMIT



Summary of Memory Operations

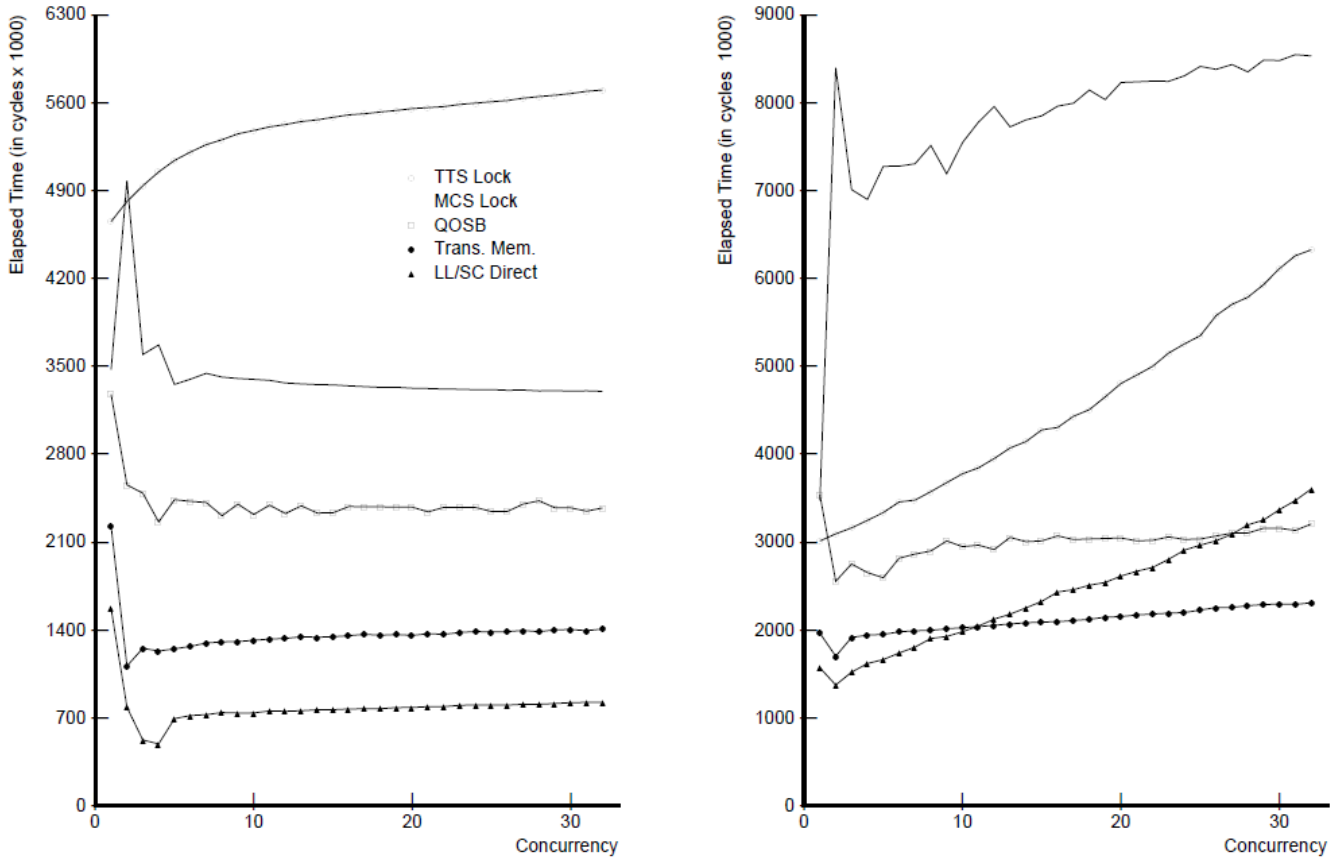
	Not owned/E	XC/XA	N
LT	issue TR bring in two lines tagged XC and XA set state to S, return data	return data stored in the XA line	change tag to XA, copy this line to a new XC line, return data
LTX	issue TRFO, bring in two lines tagged XC and XA set state to E, return data	return data stored in the XA line. If the state is S, set state to E	change tag to XA, copy this line to a new XC line, return data. If state is S, set state to E
ST	issue TRFO, bring in two lines tagged XC and XA, set state to E, update XA line, set state to M	XC line present: update XA line XC line absent: copy XA line to a new XC line, update XA line set state to M	same as LTX, but also update XA line and set state to M

Counting Benchmark

(more performance tests available in the paper)

```
shared int counter;

void process(int work)
{
    int success = 0, backoff = BACKOFF_MIN;
    unsigned wait;
    while (success < work) {
        ST(&counter, LTX(&counter) + 1);
        if (COMMIT()) {
            success++;
            backoff = BACKOFF_MIN;
        } else {
            wait = random() % (01 << backoff);
            while (wait--);
            if (backoff < BACKOFF_MAX)
            backoff++;
        }
    }
}
```

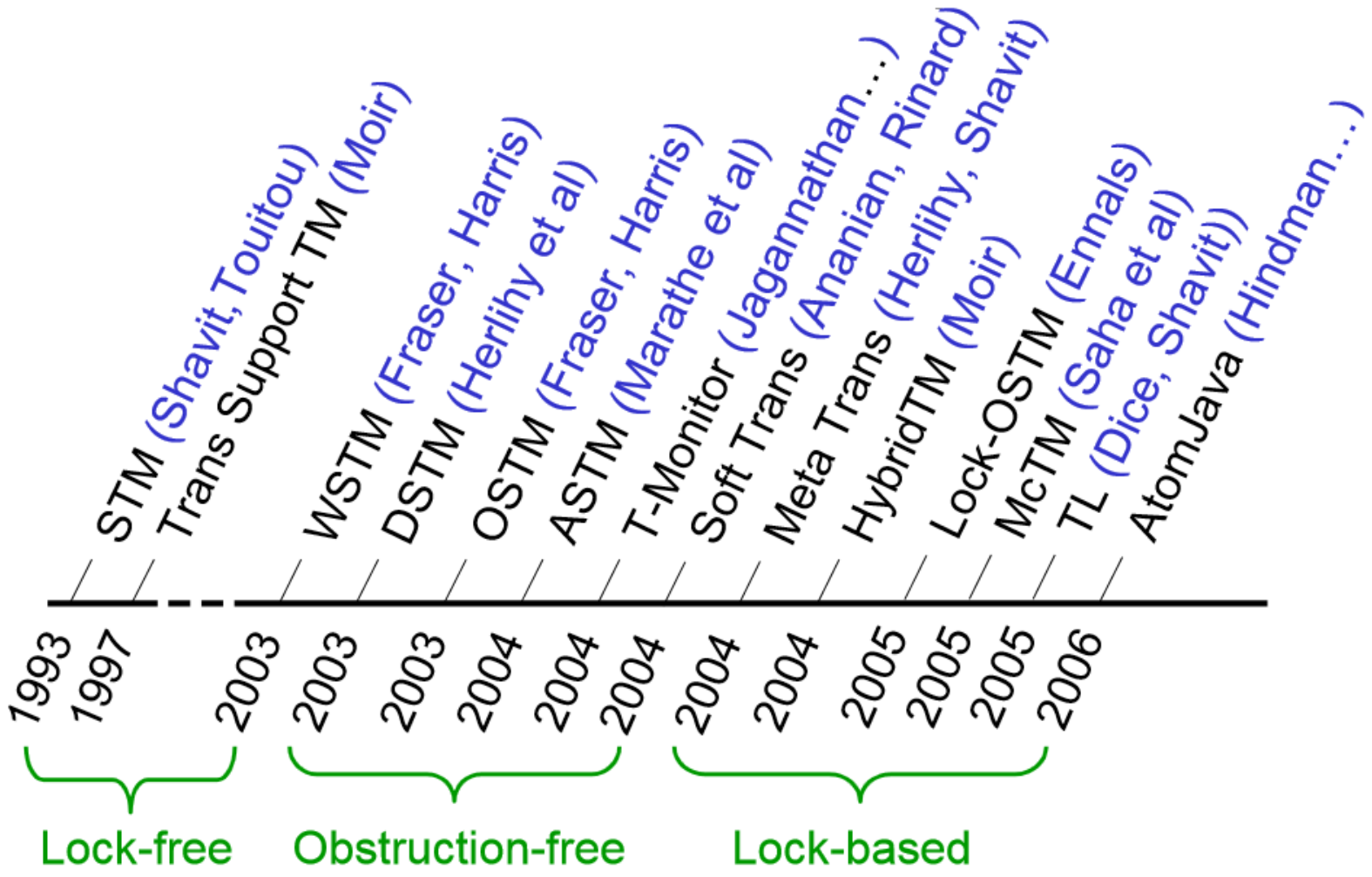


Counting Benchmark: Bus and Network

Summary of HTM

- Implemented at the granularity of cache lines.
- A special transactional cache is added in parallel to the regular cache: basically a private workspace.
- Two copies are needed in the transactional cache for modification. On committing or aborting, one of them is discarded.
- Cache lines in the transactional cache are tagged to reflect their transactional states.
- The snooping is handled similar to normal cache, except that only those lines tagged N are visible to outside processes.

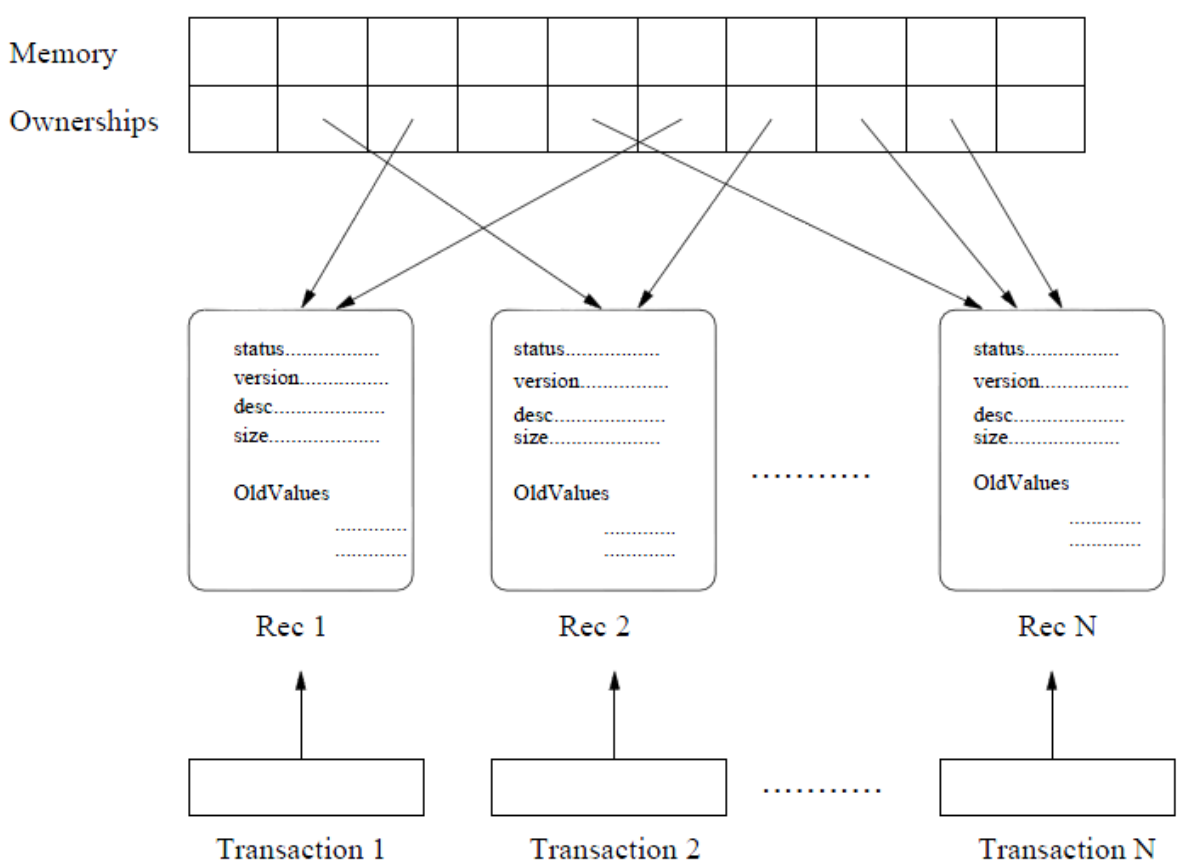
STM: a brief history



(taken from Shavit's 2011 talk)

Shavit's Original Implementation

- Before updating, a transaction needs to make a system-wide declaration to claim ownership of the shared memory words involved in the update.
- Exclusivity: each word can be owned by only one transaction at a time.
- Each word has its corresponding ownership record. Each transaction also has its corresponding transaction record of all the words it currently owns.
- Imposes total global ordering on the shared words to avoid livelock. Uses non-recursive helping on conflict.



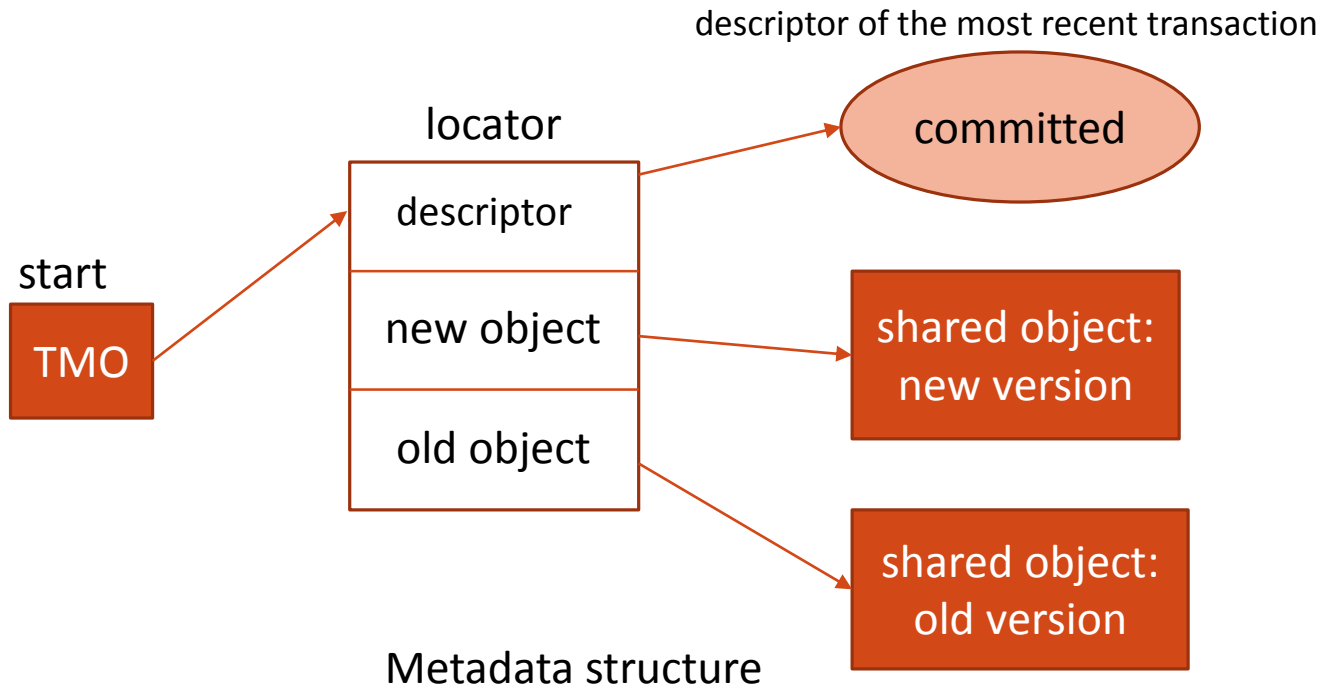
DSTM: an Object-based TM Model

Definitions in Herlihy's 2003 paper:

- Transaction: a short-lived, single-threaded computation that either commits or aborts.
- Transactional object: a container for any regular object.
- Transactions can access the contained object by "opening" it: that is, its open() member function.
- Changes to objects opened by a transaction are not seen from outside until committing.
- Commit: changes take effect; otherwise: changes are discarded.
- The implementation bears similarities to the private workspace idiom we have discussed.

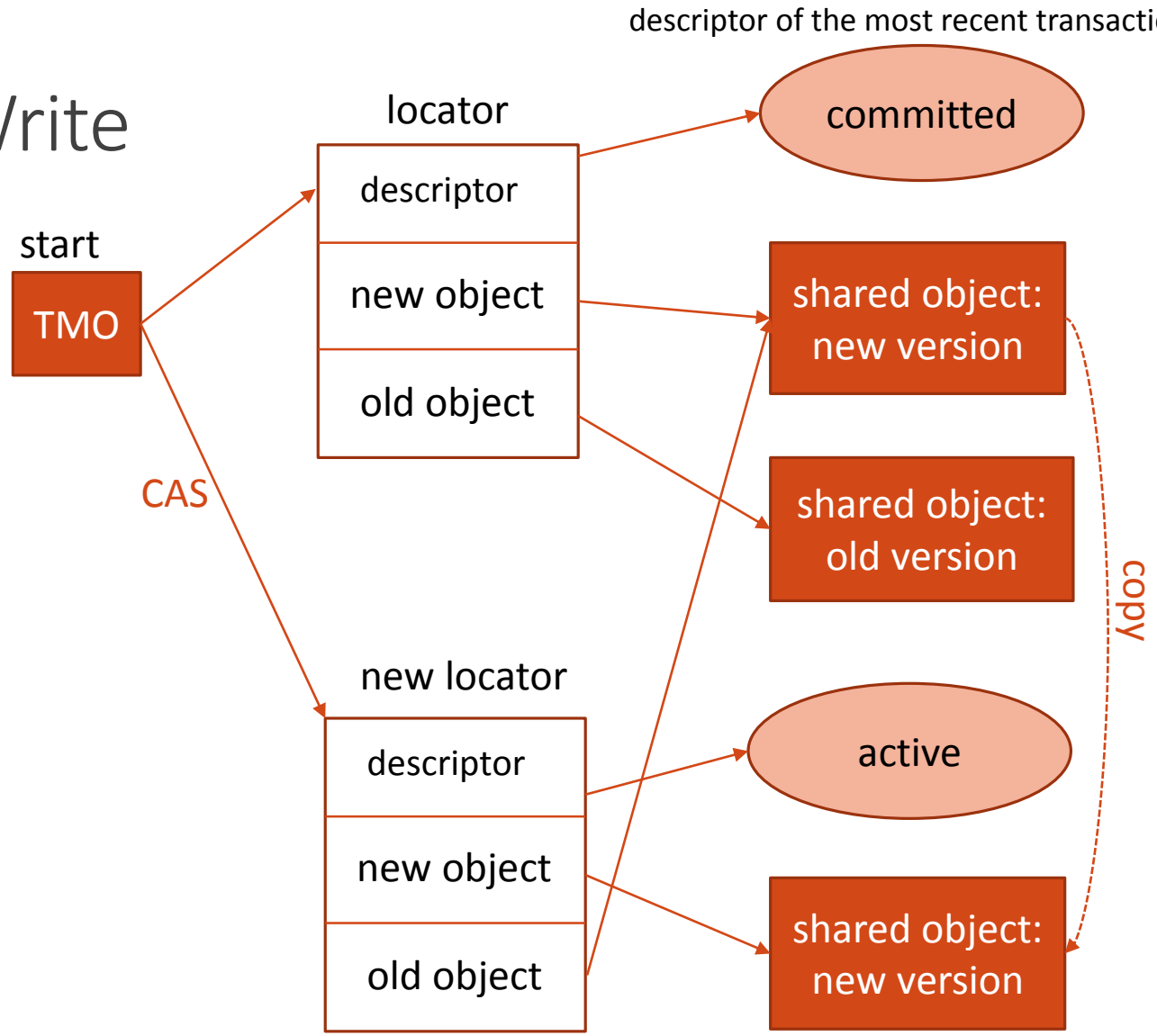
DSTM: Metadata

- DSTM is designed for concurrent access to dynamic objects.
- Obstruction-free: simplicity and flexibility.
- Contention manager to arbitrate conflicting transactions.
- Early release for contention reduction (more on this later).
- Uses eager acquire: ownership granted at access time.



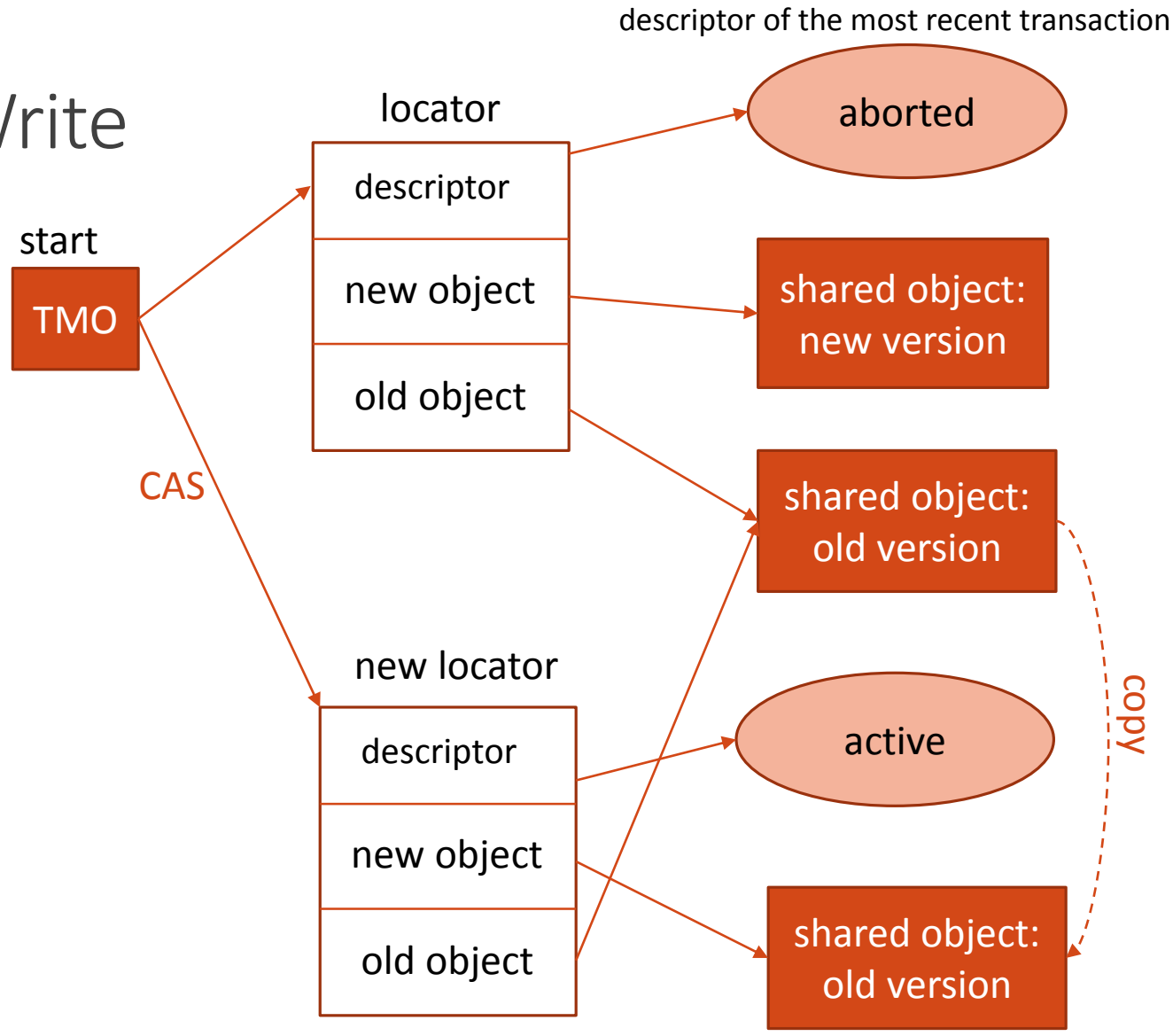
Opening a TM Object for Write

- open() is usually defined as a member function in the TM object class. Transactions need to “open” the TM object before accessing it. open() operation grants the ownership of the object (a clone) to the transaction.
- The operations are different if the most recent transaction is a commit or an abort or active.
- If the most recent transaction is committed, a copy of the new version is made, and the new locator directs its pointers to the new version and the copy.



Opening a TM Object for Write

- If the most recent transaction is aborted, a copy of the old version is made, and the new locator directs its pointers to the old version and the copy.
- If the most recent transaction is still active, we have a conflict. The contention manager (polite, aggressive, etc) will decide whether to abort. The incoming transaction invokes contention manager to make a decision.
- DSTM uses early conflict resolution. i.e. conflicts are resolved when calling open().

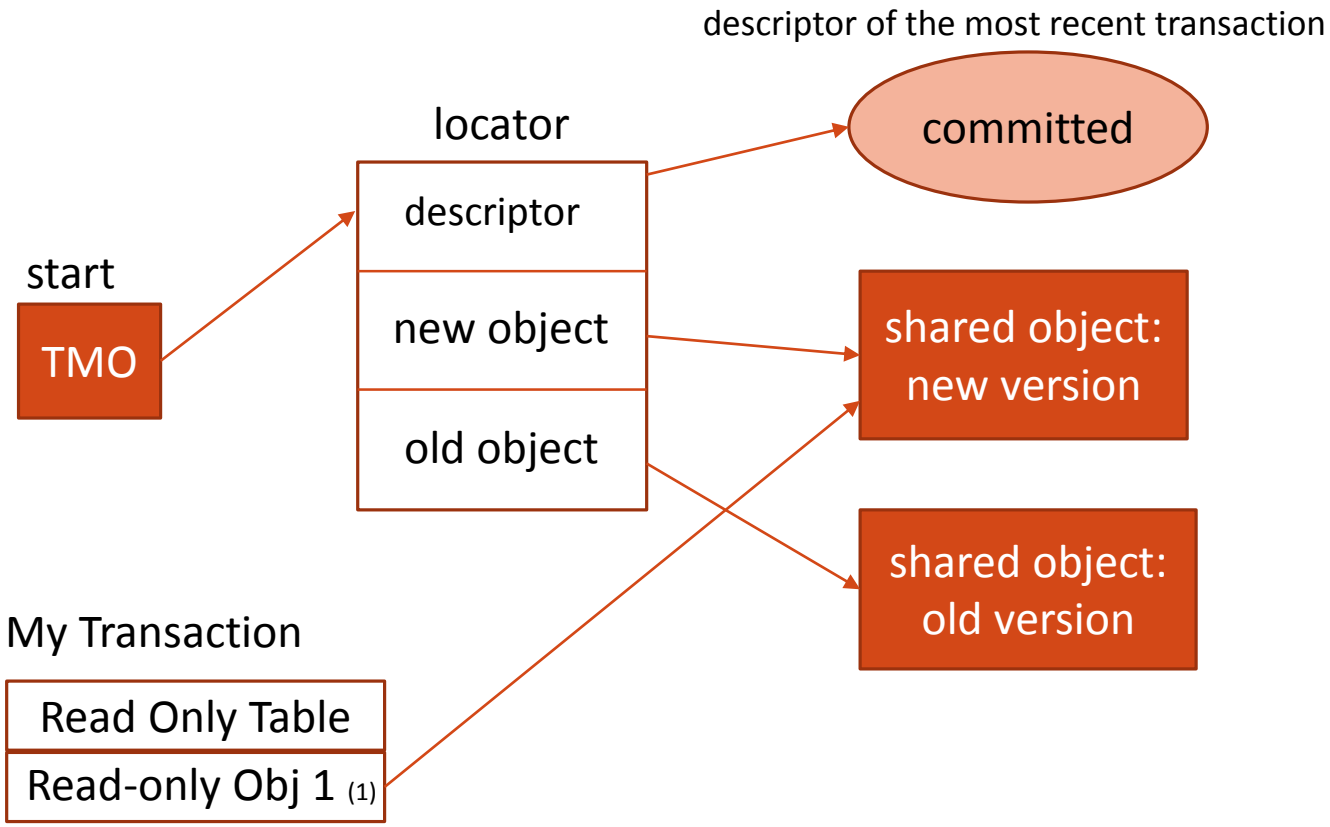


What about Reads?

Two transactions accessing the same object do not necessarily have to abort each other. Different open modes (READ, WRITE) are implemented to resolve issues like this. Need to be careful: may cause violation to isolation.

Validate: check each entry in read-only table to be the latest version, and check the status. When to perform? On every open() call and commit() call. (How to validate the write set?) This validation incurs an $O(n^2)$ overhead on the number of open reads.

On commit: first validate, then CAS the status field from ACTIVE to COMMITED.



Contention Management

Further conflict reduction is achieved by having early release: a transaction can release an object opened in READ mode even before it commits. Releasing an object before committing may cause violations to the isolation idiom (can you think of an example?).

DSTM's contention manager:

- Every call to the contention manager will eventually return a result.
- A transaction that has repeatedly requested to abort other transactions will eventually be permitted to do so (preserves obstruction-freedom and eliminating deadlocking).
- Aggressive: immediately grants permission to an abort request. Polite: adaptively backs off a few times before granting permission to an abort request.

Why Validate?

- Bank account example revisited: suppose we have two transactions: one to observe account A and B's current balance, one to transfer $50 from A to B. The transactions are shown on the right.

```
//Transaction 1

account& c1 = A.open(READ);
```

```
//Transaction 2

account& c3 = A.open(WRITE);

account& c4 = B.open(WRITE);

c3 -= 50;

c4 += 50;

commit();
```

When open() in transaction 2 is called, transaction 1 is not aborted since reads are invisible. After transaction 2 commits, transaction 1 observes a partially done transfer.

inconsistency

```
account& c2 = B.open(READ);
```

Same example can be used to show why early release may cause inconsistency.

Acquire/Read Policy

- Eager acquire or lazy acquire?
- Eager acquire: attempts acquire on first access. Avoids transactions that are doomed to abort. But the initiator may fail to commit itself, thus may victimize an otherwise valid task.
- Lazy acquire: attempts acquire before commit. Avoids unnecessary conflict detections. But may waste resource on a transaction that is doomed to fail.
- Visible or invisible reader?
- Invisible reader: transaction can open objects in read-only mode (in a local table), and determine which version to read (can see the writer). But writer do not know there is a reader currently reading. Repeated validation is costly.
- Visible reader: writers can see transactions that are currently reading a shared object, thus are able to make decisions of whether to abort itself or abort the readers or ignore. Avoids repeated validation cost.

?

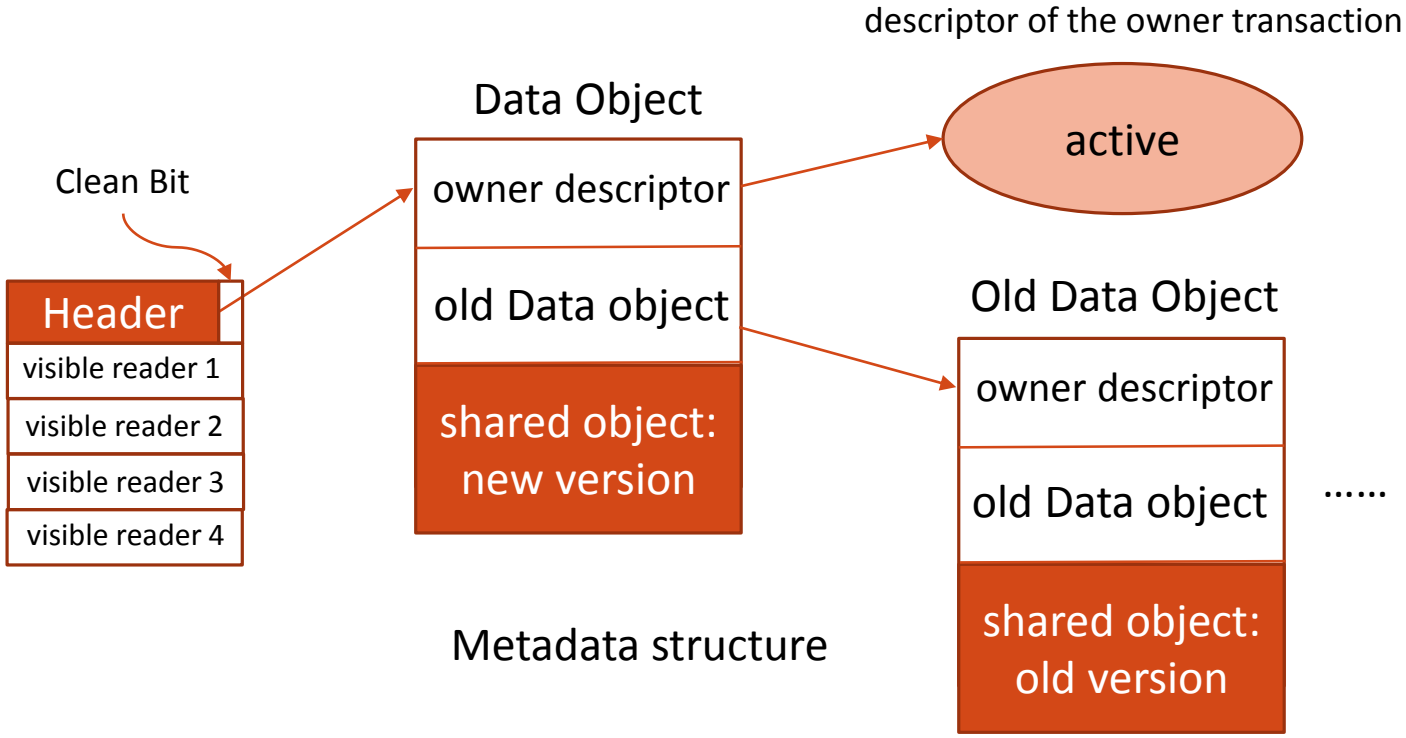
What type of policy is implemented in DSTM (just discussed)?

RSTM

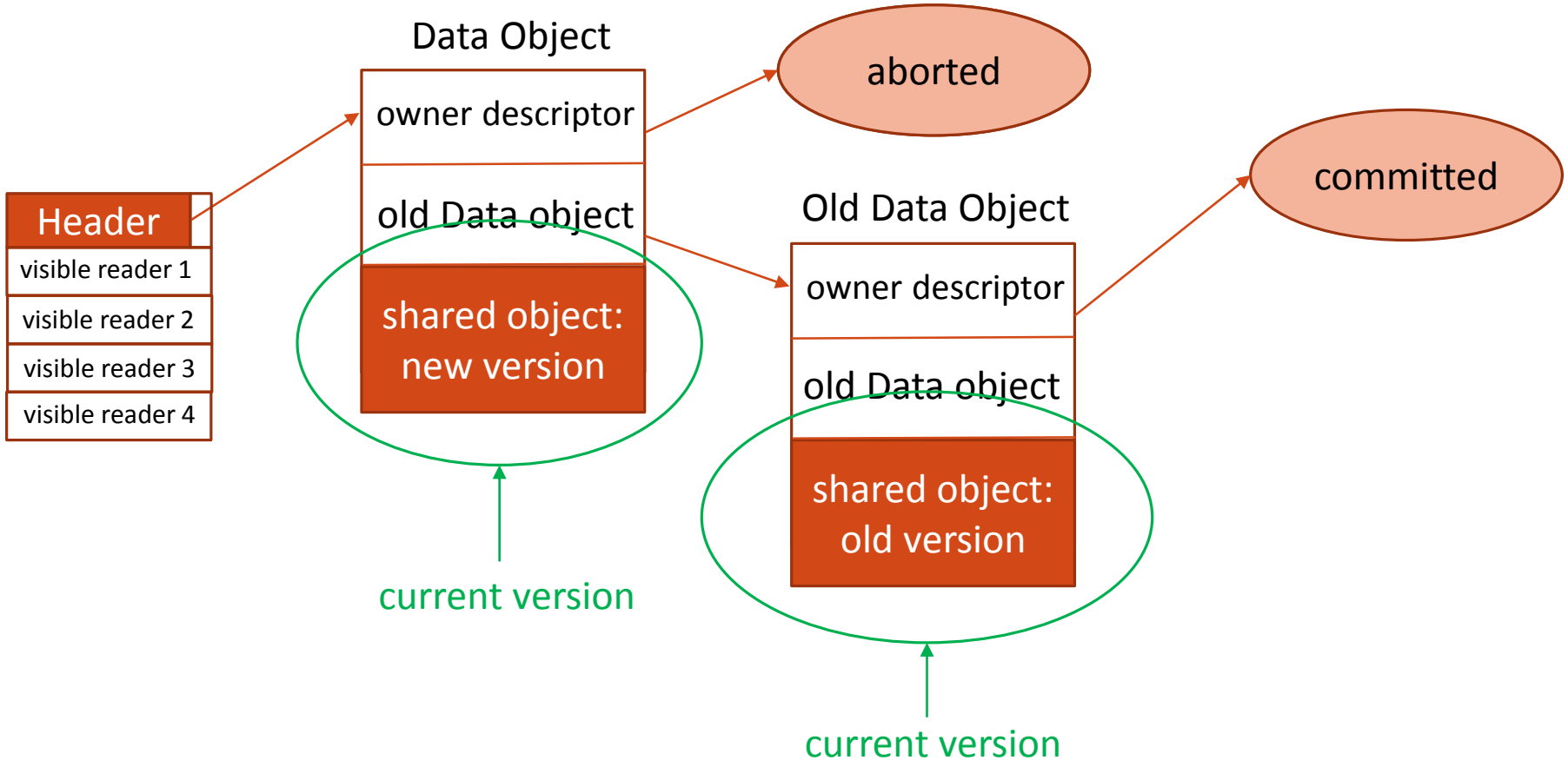
- A novel organization of metadata abstraction: only one level of indirection
- Avoids dynamic allocation of anything other than copies of data objects.
- A lightweight, epoch-based collector.
- A lightweight heuristic for visible reader management.

RSTM Metadata

- A writer allocates a new data object, and points an object header to it.
- In the data object, “old object” still references old version of the object, new version is encapsulated within the data object, instead of an indirection.
- Clean bit: 0 if the data object is current. 1 if there is (once) an outside owner. If owner is committed, then the encapsulated new version is current. If owner is aborted, then the old version is current.
- The object keeps a visible reader list to track which transactions have opened it in READ mode.



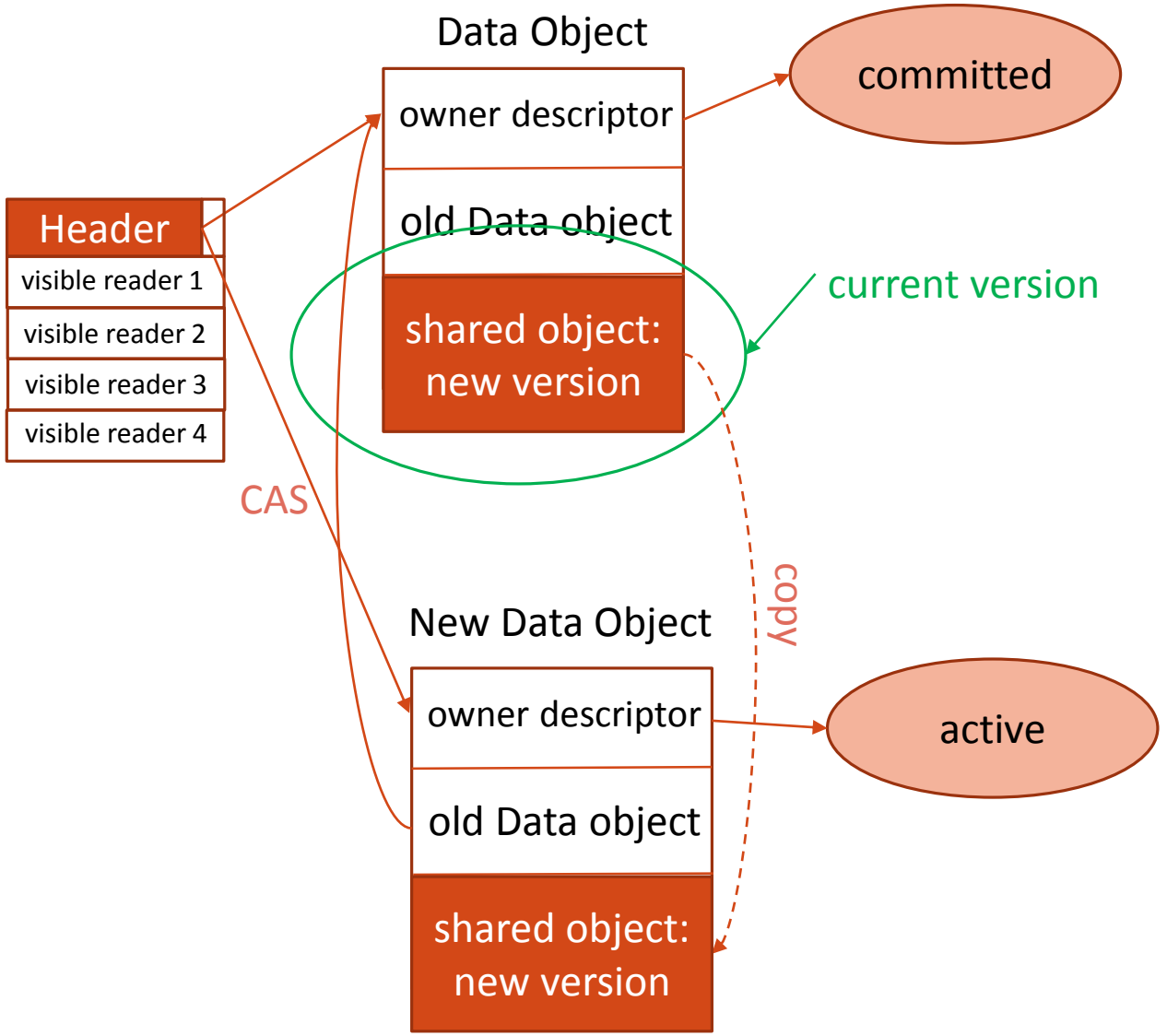
RSTM: Metadata



RSTM: Eager Acquire

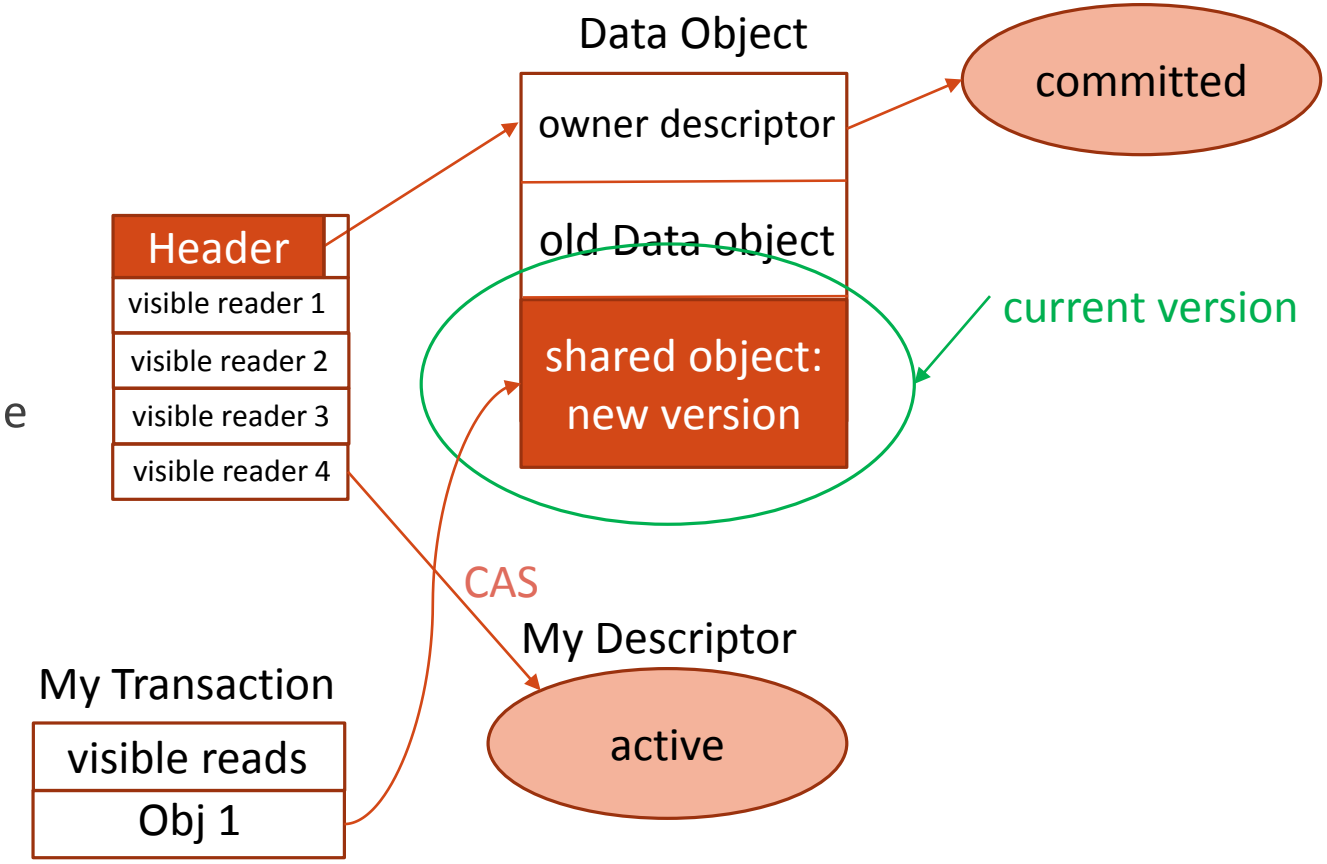
To effect an eager acquire, a transaction should:

- Reads the object header pointer.
- Finds out the current version.
- Allocates a new data object, copy old data object.
- Sets descriptor and old data object pointer.
- CAS the object header.
- Adds an entry to the private write list.
- Aborts all visible readers associated with this object.



RSTM: Reads

- A transaction can either visibly or invisibly read an object. Each transaction keeps two read lists: one for visible reads, one for invisible reads.
- Reads data object pointer and identifies the current version.
- Searches through the object's visible reader list for an empty slot. If successes, it CASes the slot pointer with a pointer to its own descriptor.
- Double check (why?) the header pointer to make sure there is no new writer coming in, then adds the object to its visible read list.
- If no empty slot, it adds the object to its invisible read list.



Opening an Object

- Acquire the object in READ or WRITE mode (by calling open_ro() or open_rw() member functions), using the methods just described.

- Add this object to either of the following three private lists: write, visible read or invisible read.

- Validate the previously opened objects.
  - If the object reference is in the write list, or the visible read list, simply check the transaction's descriptor.
  - For objects in invisible read list, validate all of them one by one, using methods described earlier.

Finishing a Transaction

- The thread traverses its write list. For commits, the thread simply tries to CAS the clean bit from 1 to 0.
- For aborts, the thread tries to CAS the header pointer from the dirty reference to an older clean reference.
- If CAS fails, then either the header pointer has changed due to an incoming acquisition, or the object is already cleaned up by another thread.
- The out-of-date data object is then marked for reclamation.
- Once the thread finishes traversing its write list and has cleaned up all related objects, its descriptor can be reused for a new transaction.
- For reads, the thread traverses its visible read list, clean up the destination slots pointed to by each entry.

Storage Management

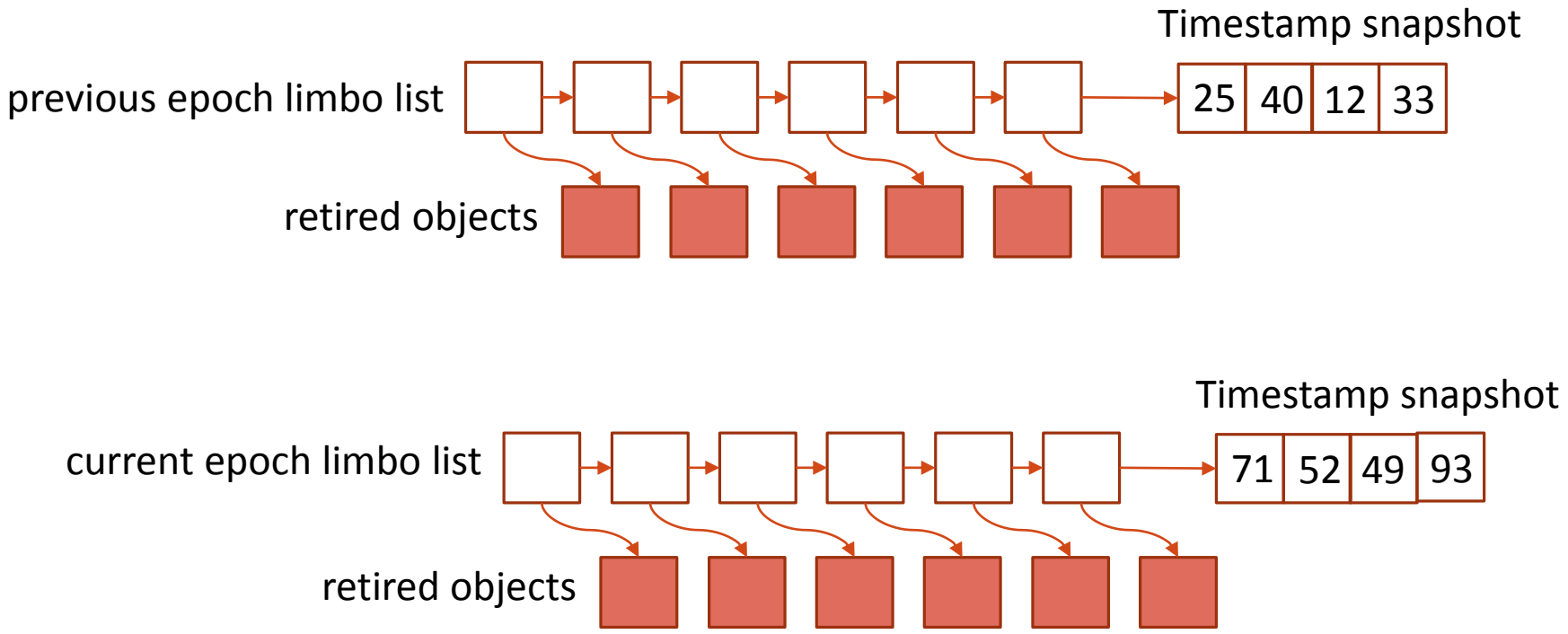
- RSTM requires dynamic allocation for copies of data objects. How are these objects managed? When a transaction is cleaning up, it knows which out-of-date data objects need to be deleted.

- When cleaning up, thread may observe out-of-date object allocated by someone else. Instead of direct deletion, the clean up thread marks this out-of-date object as "retired" and add it to its creator's "limbo list". Therefore each thread's own limbo list is always consisted of retired objects created by itself.

- Each transaction is assigned a global serial number (incrementing). A global timestamp array is maintained to keep track of the serial number of transactions currently run by all threads.

- Each thread periodically takes snapshots of the timestamp array, and attach it to its limbo list. Once this is done, an epoch is defined (the limbo list with a timestamp array attached). It then check all the limbo lists from previous epochs. A previous limbo lists is deleted if it is dominated by the current epoch (from the attached timestamp array). Finally, it starts a new limbo list for the next epoch.

Limbo List Reclamation



Summary: DSTM and RSTM

- Atomicity guaranteed by: write ownership; Isolation guaranteed by: write ownership, visible reader or invisible reader + validation

- DSTM (obstruction free):
  - Two levels of indirection. Locator only store references.
  - READ and WRITE open modes.
  - Using invisible reader (added visible reader in later implementations) and eager acquire implementations.
  - Storage management handled by garbage collector.
  - Early release for contention reduction.

- RSTM:
  - One level of indirection. The data object actually store new version of the data.
  - READ and WRITE open modes.
  - Both invisible and invisible readers, and both eager and lazy acquire (not shown in this talk) implementations.
  - Storage management handled by epoch-based limbo list.

Summary: Contention Detection Pros and Cons

Eager vs Lazy Acquire

- Eager: avoids transactions that are doomed to be aborted wasting resources.
- Lazy: avoids transactions being aborted by another transaction who aborts later.

If a transaction has many read-only opens and few read-write opens, which strategy should be preferred?
What about a transaction with opposite properties?

Invisible vs Visible Read

- Invisible: reduces contention for read-only opens, but increases validation overhead.
- Visible: reduces validation overhead, but increases contention by writing to object metadata even for read-only opens.

If a transaction has many read-only opens, which strategy should be preferred?
If the underlying algorithm has a highly contended hotspot, which strategy should be preferred?

References

Transactional memory: architectural support for lock-free data structures, Maurice Herlihy, J. Eliot and B. Moss (May 1993) In: *ISCA '93 Proceedings of the 20th annual international symposium on computer architecture*, Pages 289-300

Software Transactional Memory, Nir Shavit and Dan Touitou (Aug 1995) In: *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. pp. 204--213.

Software Transactional Memory for Dynamic-Sized Data Structures, Maurice Herlihy and Victor Luchangco and Mark Moir and III William N. Scherer (Jul 2003) In: *PODC~'03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*. pp. 92--101.

Advanced Contention Management for Dynamic Software Transactional Memory, William N. Scherer III and Michael L. Scott (Jul 2005) In: *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*. Las Vegas, NV.

Lowering the Overhead of Software Transactional Memory, Virendra J. Marathe and Michael F. Spear and Christopher Heriot and Athul Acharya and David Eisenstat and William N. Scherer III and Michael L. Scott (TRANSACT 2006, Ontario, Canada)

A Qualitative Survey of Modern Transactional Memory Systems, Technical Report Nr. TR 893. Computer Science Department, University of Rochester

What Really Makes Transactions Faster? David Dice and Nir Shavit (Jun 2006), In: *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*

Conflict Detection and Validation Strategies for SoftwareTransactional Memory, Michael F. Spear and Virendra J. Marathe and William N. Scherer III and Michael L. Scott (Sep 2006) In: *Proceedings of the Twentieth International Symposium on Distributed Computing*.

Thank You!